NORTH-HOLLAND

A PLATFORM FOR RESTRICTION MAPPING

CHUT N. YEE* AND TREVOR I. DIX

▷ Restriction mapping involves finding certain sites on a DNA molecule. Biologists perform a variety of experiments to obtain data that are subject to error of method. The diversity of experimental data makes restriction mapping a complicated synthesis of information processing and constraint satisfaction. We present a theoretical framework that treats constraints for the various techniques in a homogeneous and consistent manner. Our implementation is written in CLP(R), includes the main techniques used by biologists, and encourages the building of maps incrementally. We also show how to consider more recent experimental techniques in the framework. © Elsevier Science Inc., 1997 ◁

1. INTRODUCTION

The inherited characteristics of an organism are contained in its genome, a large deoxyribonucleic acid (DNA) molecule. The coding is in terms of the nucleotide building blocks or *bases*. At an abstract level, there are four bases A, C, G, T. DNA may be viewed as a linear sequence of bases. To give an idea of scale, consider the current Human Genome Project, which has as a primary goal to identify and sequence all genes in the human genome. The genome is about 3×10^9 bases, and genes vary by roughly an order of magnitude about 10^3 bases. There is a practical need to break DNA molecules down into smaller fragments. For example, current sequencing machines process about 500 bases before reaching an unacceptable error rate. A common method of representing fragments is in a restriction map, which is derived from experimental data.

*Supported by Australian Research Council grant A49330684 and a Monash University FCIT grant.
Address correspondence to Trevor Dix, Department of Computer Science, Monash University, Clayton, Victoria 3168, Australia, Email: trevor@cs.monash.edu.au.
Received December 1994; accepted September 1996.

655 Avenue of the Americas, New York, NY 10010
0743-1066/97/$17.00
PII S0743-1066(96)00118-5

A particular *restriction enzyme* recognizes each instance of a particular pattern (again in terms of a sequence of bases) in the DNA, and *cleaves* (or *digests*) at all such points, producing *fragments*. Patterns are typically four, six, or eight bases in length. Since the shorter patterns occur more often, these enzymes tend to produce more fragments than those recognizing longer patterns.

A *restriction map* specifies the sites at which particular restriction enzymes cleave the DNA molecule under consideration. A restriction map has many usages, such as enabling a biologist to extract a particular fragment for experimentation, or merging information for one fragment (its sequence, say) with information from other fragments, with which it overlaps. The map is inferred from the length of fragments produced by various digestions by restriction enzymes.

It is worth noting at this time that the DNA experiments covered in this paper deal with a solution of cloned DNA copies, although we simply refer to the DNA. Any anomalies are in effect swamped by the average of the other clones used. We also limit ourselves to circular DNA, the most typical form of experimental data for restriction mapping. As it turns out, linear fragments of DNA can be inserted into a smaller circular molecule. Typically, an insert of around 500 up to 1,000,000 bases can be performed. So although the DNA of interest may be linear, the experimental data come from circular DNA.

Figure 1 shows a restriction map for a circular DNA molecule. The sites are labelled with an enzyme (fictitious in this example) and a site number in a clockwise direction. Were this DNA cleaved by enzyme *a*, three different sized fragments would result: one from site number 1 to number 4 (in a clockwise sense), another from 4 to 6, and another from 6 to 1. This is an example of a complete *single digestion* (SD), where one enzyme is given sufficient time so that all *a* sites in all clones are cleaved. A complete SD with enzyme *b* would result in two fragments: one from site 2 to site 7, the other from 7 to 2. One can readily see where the *a* fragments overlap with the *b* fragments in the map and how information could be merged. For example, if from a separate subexperiment for the large fragment from site 2 to site 7, one were able to find the relative positions of the two *c* sites, merging would result in the map as shown.

After a digestion, the lengths of the fragments are estimated. This is performed in a gel electrophoresis experimental procedure: the solution of all fragments is placed at a point (in a well) on a plate of polymer gel; a potential is applied for a fixed period of time; the charged fragments migrate a distance roughly propor-

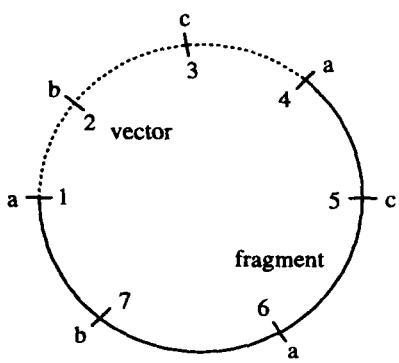


FIGURE 1. Circular restriction map with site for three enzymes.

tional to the logarithm of their mass, forming bands corresponding to different fragment sizes. On the same gel and at the same time, but in a different well, a set of standard fragments (whose lengths are known exactly) are migrated. From the standard, the lengths of the other fragments are estimated. Errors are introduced in this method.

Returning to the fragments from the SDs above with enzymes *a* and *b*, the two separate sets of information provide no information for the relative site positions. One approach to gain this information is to perform a separate complete double digestion (DD) with both enzymes *a* and *b*. In this case, five fragments would result from sites 1 to 2, 2 to 4, 4 to 6, 6 to 7, and 7 to 1. By permuting each of the three sets of fragments, a map can be found where sites align consistently. In practice, a set of maps can be found within the bounds of experimental error. Reflection and rotation can be ignored.

This is a typical DD experiment. It can be generalized for more enzymes and more pairs, for example, a SD with enzyme *c* and a DD with enzymes *b* and *c*. There are many ways in which information from separate DD experiments are merged.

The process for inserting linear DNA into circular DNA can be used for any subfragment, thus breaking a problem down to smaller subproblems. Submaps resulting from subsequent subexperiments can be merged into a parent map. For the *a* and *b* enzymes DD experiment, it is not possible to obtain a total order for sites 4 and 6 without additional information. One way to resolve the problem is to perform a DD experiment with enzymes *a* and *c* in a subexperiment on the fragment from site 2 to 7 (inserted into an appropriate vector). The merging of the *a* sites and the *c* sites from the subexperiment would then specify the map.

Generating a restriction map is essentially a permutation problem. To help limit exponential explosion in the search, both manually and with computer assistance, biologists use additional information. One way is to uniquely label a point on the DNA, effectively tying fragments with that label to that point. This form of labelling is called *probing*.

Other obvious information to exploit is knowledge of the circular DNA into which the insertion was made. In Figure 1, the linear fragment of interest (from site 4 to site 1) has been inserted into another small circular DNA molecule, generically called a *vector* (from site 1 to 4). The insertion is done at a known restriction site (for enzyme *a* in this case). Here the vector has one site for each of enzymes *a*, *b*, and *c*. The insertion was made at the *a* site. If the inserted fragment did not originally have *a* sites at each end, it is possible to manufacture such sites. The biologist has many different vectors to choose from; the sites within vectors are known precisely. Provided the vector fragments can be distinguished from the other fragments, for example because of unique fragment lengths, part of the map is already known (which we address in Section 6).

End fragments, which contain part of the vector, do not correspond to a single fragment in the vector or in the inserted DNA. Some SD fragments with enzymes *b* or *c* in Figure 1 illustrate the problem. For example, for enzyme *c*, the fragment from site 3 to 5 contains part of the vector between sites 3 and 4. Sometimes biologists deliberately label such fragments. In effect, a label is attached to a specific part of the vector (between sites 3 and 4 in this case) using knowledge of the sequence of bases. Fragments containing this part of the vector can now be identified. This process is called *end-labelling*.

The techniques mentioned above cover a wide spectrum of approaches used by biologists to obtain data to construct restriction maps. In this paper, we show how all such data can be merged using constraint logic programming to provide an appropriate platform. Although we introduced the problem by referring to the Human Genome Project, our goals are much more modest. We aim at small restriction maps with, say, 5 to 10 submaps. Such sized problems are very common and provide a basis to concatenate linear maps.

The treatment of error in experimental data in this paper is similar to most literature in the area and to that used by biologists in calculations by hand. Each fragment has an associated error bound. If, for example, two DD fragments correspond to the parts of a SD fragment, then the bounds of the DD sum should lie within the bounds of the SD fragment. This naturally leads to a system of linear inequalities. However, with very few exceptions, most authors treat the inequalities locally rather than as a system.

A constraint logic programming language such as CLP(R) [11, 12] provides a built-in linear programming solver and suitable symbolic manipulation for this type of system of inequalities. Indeed, our work illustrates the expressiveness and flexibility of constraint logic programming. To implement a system for analysis of data from different experimental techniques and combine resulting maps would be a daunting task in a conventional imperative programming language. However, CLP(R) eliminates the need to program a specialized inequality solver and, by extensive use of logical variables and partial instantiation, provides appropriate mechanisms for the task. The system was developed incrementally, adding code for different techniques. Moreover, the system encourages combining information in the same fashion in which experimental data is produced.

Some of the material in this paper is found in Dix and Yee [6], where techniques for the merging of maps are introduced. Experimental results also are given therein and shown to be highly data-dependent. This paper does not assume so much biological background material. We give more code and show that the integration of data from a wider range of experiments is not difficult, deriving constraints for several additional techniques. We contribute several optimizations in our operators for merging maps.

Having introduced the problem and motivated the use of constraint logic programming, we now outline the rest of the paper. We start with the representation of maps both graphically and by constraints. Since there is no similar work in terms of generality of digestions and merging maps that would enable comparisons to be made, Section 3 mentions related work in the field. Our approach and permutation generation is then demonstrated for handling SD and DD data in Sections 4 and 5. Section 6 describes the use of vector information, and Section 7 presents the constraints and operators for merging maps. Before concluding, we develop constraints for additional techniques and discuss their use.

2. REPRESENTATION OF RESTRICTION MAPS

Biologists draw maps such as that in Figure 1 that usually have distances between the sites as well. We represent restriction sites as points on a real line and digest fragments as distances between sites. The lengths of fragments are not measured exactly. A fragment *constrains* the distance between two sites within the experimental error bound.

Single- and double-digest fragments can be shown as a graph [2], where each line represents either a SD or DD fragment joining two sites. For example, the graph for Figure 1 is shown in Figure 2. The map is opened up at site 1; sites 1, 2, and 3 are duplicated as 1′, 2′, and 3′, respectively, to represent wraparound; and the map for the digestion for enzyme *a* is duplicated to obtain a planar diagram. Generally, horizontal lines represent SD fragments, for example between sites 1 to 4. Diagonal lines represent DD fragments, for example between sites 1 and 2. Adjacent parallel lines indicate more than one fragment joins the two sites. For example, sites 4 and 6 have one SD fragment and one DD fragment joining them.

Digest[E] denotes a list of fragments from a digestion by the enzymes in the list E. So Digest[a] is a SD with enzyme *a* and Digest[a, b] is a DD with enzymes *a* and *b*. Fragments from a digestion are denoted by the enzyme, or enzymes, and a fragment number, as in a1, b2, ac1, etc.

Like most of the literature, we treat experimental error as a discrete error bound. The bound is often taken as ± percentage of the fragment length. Experimental error in the length of a fragment is represented by a lower and an upper bound. The distance between two sites is constrained by the placement of a fragment between them.

Map generation proceeds from left to right in the diagrams. If a fragment is placed between sites s_i and s_j, $j > i$, the site constraints are

$$L \leq s_j - s_i \leq H,$$

where H and L are the upper and lower bounds of the fragment concerned. When $i > j$, the fragment spans the circular boundary, and the site constraints become

$$L \leq s_j - s_i + C \leq H.$$

C is a variable that denotes the length of the circular map [4].

For a solution map, all constraints imposed by all fragments must be satisfiable. There may be a set of consistent maps each falling within the error bounds or having some fragments permuted because the data only provide sufficient information to obtain a partial order. We wish to find all such consistent maps.

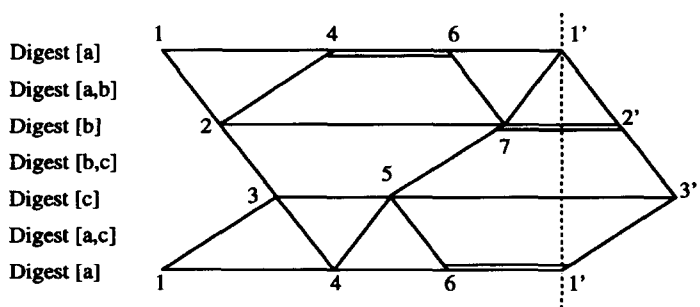


FIGURE 2. Opened map.

3. RELATED WORK

A summary of many of the experimental techniques can be found in Zehetner et al. [17]. The number of fragments, number of enzymes, and experimental error make finding restriction maps difficult. Goldstein and Waterman [8] point out this is an NP-complete problem. Indeed, the desire to manage the complexity is part of the reason biologists resort to other supplementary experimental techniques. Generally, these reduce the permutations to be considered, or produce maps for individual fragments, which can be combined later. Maps are usually built from one end by permuting fragments, testing bounds on fragment lengths, and backtracking when necessary.

As mentioned earlier, our representation results in a system of linear inequalities, where the constraints on sites for one fragment may affect any other site. However, early work on restriction mapping treated constraints imposed by fragments only as local constraints. To illustrate, local constraints for the DD fragment between sites 2 and 4 in Figure 2 would only affect sites 1, 2, 4, and 5. The main developments employing local constraints were by Stefik [15], Pearson [14], Fitch et al. [7], Zehetner and Lehrach [18], and Krawczak [13].

Allison and Yee [2] recognized that the restriction mapping constraints fall into separation theory. Hence, constraints are global and indirectly impose constraints on other fragments. Allison et al. [1] developed a fast algorithm applicable to linear restriction mapping that was applied by Ho et al. [10]. Dix and Ho-Stuart [4] developed phased separation theory for circular DNA.

Allison and Yee [2] and Bellon [3] used logic programming for the RSM problem. Yap [16] demonstrated the use of CLP(R) for experimental data from two single-digests and one double-digest like that mentioned in the Introduction. In unpublished work prior to that, Yee reprogrammed his early work in CLP(R).

4. OUR APPROACH

Being a combinatorial problem, restriction mapping is computationally intractable for large numbers of enzymes and fragments. Even for problems of a relatively small number of fragments, the number of solutions within the error bounds is often too big to be useful [9]. To facilitate the mapping, biologists usually resort to combining maps obtained by analysis of data from supplementary experiments such as subexperiments on an extracted SD fragment, partial digestion, hybridization, and end-labelling. Each of these techniques provides a significant reduction in the search space and solution space for mapping. They are becoming indispensable in the drive to solve bigger problems. Stochastic methods offer the only hope for large problems. However, for the above techniques, stochastic algorithms have been relatively ineffective.

The mapping process is incremental—the next experiment being decided by analysis of and reasoning about the currently available experimental data. Unlike the previous approaches, our approach recognizes the fact that data are typically not all available at once. Just as biologists resolve ambiguities and reduce time complexity by solving subproblems, we aim for a platform that supports the combination of information from subproblems to solve larger problems.

The diversity of techniques mentioned above is nontrivial. Prior to our work, little attempt has been made to address these diverse and complicated additional

techniques in a uniform and consistent manner. Most mapping programs solve only simple problems. As experimental techniques evolve, different data analysis routines need to be added to mapping programs. Our experience in using the C language for mapping problems suggests this is a daunting task.

We see constraint logic programming as an ideal tool to tackle the problem. We have found CLP(R) provides flexibility that enables us to accommodate new techniques with very few changes to the underlying map generator. We have found that existing low-level code can be reused; only new code for the new technique is added.

Our system provides: a generalized simultaneous map generator, methods for combining maps from subexperiments on subfragments, pipelining and cross multiplication schemes for combining maps, and consideration for vectors. These operators have been reported in Dix and Yee [6]. Here we give more code (down to a level where a logic programmer would have no difficulty imagining the code) and we give the constraints to illustrate how constraint logic programming enables us to integrate the methods consistently. We also show how to incorporate other techniques into the CLP(R) based platform by developing the constraints for methods dealing with labelled fragments, end-fragments, and specialized partial digestions.

The main benefit constraint logic programming gives us is the automatic maintenance of constraints. In particular, CLP(R) [12] has a general constraint solver in the real-number domain that efficiently supports the incremental introduction of new constraints with its built-in linear programming solver. Coupled with backtracking on failure, CLP(R) allows the maps to be built incrementally. Moreover, there is no need to develop special-case theories from separation theory. The constraint system handles everything.

5. MAP GENERATION

Restriction mapping is a permutation problem where the search for solution maps involves permuting enzyme sites and digest fragments [15, 7, 5]. Using a recursive search, the current map is *extended* one site at a time. At each step, the site is conjectured to be for one of the enzymes in turn. For each conjecture, we iterate through all possible combinations of digest fragments that could confirm the current site.

We denote a restriction site by $s_i = (i, e_i)$, where i is the site number and e_i is the cutting enzyme. We overload this definition when referring to CLP(R) to be $s_i = (V_i, e_i)$, where V_i specifies the position of the site and is called the *site variable*.

Figure 3 shows a successful order of fragment placement for the map in Figure 1. The extension process starts from site number 1. The wraparound sites, 1′, 2′, and 3′, are necessary for placing the remaining fragments. For convenience, we have labelled fragments according to their order in the solution for a particular digestion. Table 1 illustrates the successful placement of fragments as the search proceeds (without backtracking). At each successive site, the associated enzyme, site variable, and fragment(s) placed are given.

The following recursive code performs map extension. Backtracking occurs when no digest fragment can be found to confirm a conjectured enzyme for the

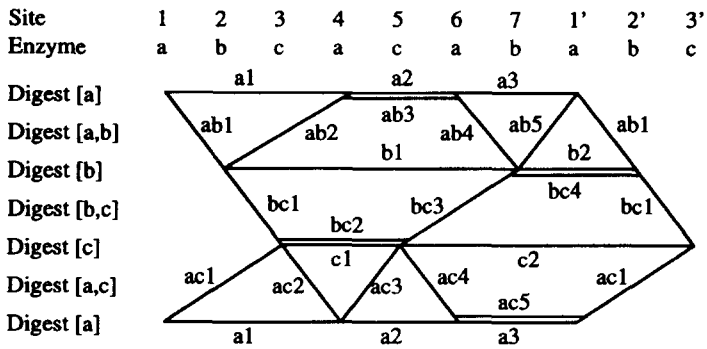


FIGURE 3. Example generation.

current site. A consistent map is found when all digest fragments are placed:

```
extend([ ], _, _, Map).
extend(Digests, Enzymes, [CurrentSite | Sites], Map) :-
  CurrentSite = (Vi, Ei),
  member(Ei, Enzymes),
  placeFragments(CurrentSite, Digests, UnusedDigests, Map),
  extend(UnusedDigests, Enzymes, Sites, Map).
```

The *extend* predicate implements the permutation generator. The first argument is the list of digest fragments to be placed; the second is the list of enzymes involved; and the third argument is the list of sites. The fourth argument is the data structure for the solution map.

The *member* goal assigns an enzyme to the current site from the list of enzymes. On backtracking, it will assign each element of the list in turn. If the enzyme is already known, for example a vector site (see Section 6), this predicate will succeed only once.

The *placeFragment* goal attempts to place fragments for the current site. The fragments are selected from *Digests* and the unused fragments are returned in *UnusedDigests*. A solution is found if the list becomes empty. This predicate will be described in detail in the next subsection.

TABLE 1. Fragments placed at each site

Site number	1	2	3	4	5	6	7	1'	2'	3'
Site variable	V_1	V_2	V_3	V_4	V_5	V_6	V_7	$V_1 + C$	$V_2 + C$	$V_3 + C$
Enzyme	*a*	*b*	*c*	*a*	*c*	*a*	*b*	*a*	*b*	*c*
Digest[a]				a1		a2		a3		
Digest[b]							b1		b2	
Digest[c]					c1					c2
Digest[a, b]		ab1		ab2		ab3	ab4	ab5		
Digest[b, c]			bc1		bc2		bc3		bc4	
Digest[a, c]			ac1	ac2	ac3	ac4		ac5		

The *allmaps* predicate below illustrates how we force CLP(R) to backtrack to find all solutions, with a failure loop, and where output is performed:

```
allmaps(Digests, Enzymes) :-
  extend(Digests, Enzymes, Map, Sites), storeSolution(Sites, Map),
  fail.
allmaps(_,_).
```

For our purposes, a solution *Map* is uniquely defined by the *site configuration* and *fragment configuration*, normalized relative to rotation and reflective symmetry. From a given starting point, the site configuration specifies the order in which the enzymes cut the DNA molecule. The fragment configuration specifies the order in which fragments are placed for each of the digests. The order in which the fragments in different digests are placed relative to each other is implicit in the site ordering. The *Map* for the example in Figure 3 is defined by

Site:	[a,b,c,b,a,c]
Digest[a]:	[a1,a2]
Digest[b]:	[b1,b2]
Digest[c]:	[c1,c2]
Digest[a,b]:	[ab1,ab2,ab3,ab4]
Digest[b,c]:	[bc1,bc2,bc3,bc4]
Digest[a,c]:	[ac1,ac2,ac3,ac4]

Vector constraints prevent the generation of reflected and rotated maps.

5.1. *Placement of Fragments*

The algorithm for placing digest fragments to confirm a new site s_c (*CurrentSite* in the call) is as follows:

```
foreach Digest[E] ∈ D do
  if e_c ∈ E then
    s_p = previousSite(E)
    if s_p ≠ NULL then
      selectAndPlaceFragment(Digest[E], s_p, s_c)
    endif
  endif
endfor
```

The first line iterates E over all digestions in the domain. The condition in line 2 states that a fragment from Digest(E) may be required only if the enzyme for the current site, e_c, is in the digestion E. The function *previousSite* searches backwards (i.e., to the left in Figure 3) through the confirmed sites for the previous site s_p for digestion E. If such a site exists, the *selectAndPlaceFragment* function nondeterministically chooses a fragment from Digest(E) and places it between the sites s_p and s_c. On backtracking, the function iterates through all fragments in the digest.

The fragments chosen for placement are removed from the respective *Digests* list, and the remaining fragments are returned via the *UnusedDigests* argument in the *placeFragments* call.

To illustrate the *previousSite* function, consider when site 5 is the new site in Figure 3. For both Digest[c] and Digest[b, c], s_p is site 3, thus both digestions place a fragment from site 3 to site 5. However, s_p may not exist, in particular at the beginning of map construction. We place a fragment only when such a site is found. For example, at site 1 no fragment is placed; also at site 2 we place only ab1 since there is no previous site for Digest [b].

The algorithm is general with respect to the number of digestions and enzymes. It is able to handle an incomplete set of digestions, i.e., not including all single- or double-digestions, and digestions with more than two enzymes, i.e., triple-digestions, etc., and an arbitrary number of enzymes.

6. VECTOR CONSTRAINTS

A vector is characterized by its length and enzyme restriction sites. Our system supports a database of known vectors with the following schema:

vector(Identifier, Length, RestrictionSites).

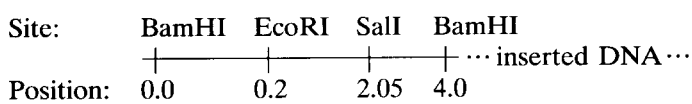
where RestrictionSites = [site(Enzyme, Position),...] specifies the restriction sites of the vector. For example, the entry

```
vector(pACYC184, 4.00, [site('EcoRI', 0), site('HindII', 1.45),
                        site('BamHI', 1.75), site('SalI', 1.95)]).
```

defines the vector pACYC184. Its length is 4k bases, and the restriction sites for enzymes EcoRI, HindII, BamHI, and SalI are based at positions 0, 1.45, 1.75, and 1.95, respectively.

The vector, being a small circular DNA molecule, has relative site positions. It is opened at the desired insertion site and the DNA fragment, which has a specific orientation, will bind to the vector. Our software adjusts the first site to be the insertion site and the others relative to it. Usually only a subset of the enzymes with known vector sites are used in an experiment. The unused sites are filtered out.

For the pACYC184 vector opened at the BamHI site and only using enzymes BamHI, EcoRI, and SalI, we have the following arrangement when DNA is inserted:



Traditionally, fragments from the vector are identified prior to the mapping process and manually removed from the data to be mapped. This approach is unsatisfactory since the vector fragments may not be uniquely identifiable within the bounds of experimental error, and removal of these fragments takes constraints imposed by the vector out of the system arbitrarily. We are able to express vector information in a consistent and homogeneous way by using the vector sites to partially instantiate the site variable before calling the map generator.

For the above example, the map generator can be called with the site variable instantiated as

Site: ['BamHI', 'EcoRI', 'SalI', 'BamHI' | _].

Where no vector sites are known, for example where there is only one vector site, no partial instantiation is made.

The vector sites are known precisely and have no error bounds. The vector constraints have the following equalities:

$$V_2 - V_1 = 0.2, \qquad V_3 - V_2 = 1.85, \qquad V_4 - V_3 = 1.95,$$

where V_1, V_2, V_3, and V_4 are site variables for sites 1, 2, 3, and 4, respectively.

Note that we place the vector sites at the beginning of the map to make maximum use of the known information. The vector sites also provide a convenient fixed reference point and automatically prevent the generation of reflected or rotated symmetrical maps.

7. OPERATORS FOR MERGING MAPS

We now present the constraints and methods used to merge maps. Merging is necessary because biologists tend not to perform all their experiments at once. Hence, not all the digest data are available at once. They tend to perform digestions with a few enzymes, trying to find a unique map. If this is impossible or if more sites are required, they proceed with further experiments, and so on. This iterative approach is partly because of the limited number of digestions that can be performed in one experimental procedure. So constructing a restriction map for many enzymes must involve several separate experiments. The maps from these smaller subproblems must be combined to form the final map. Our work is the only comprehensive approach to the problem.

7.1. Subexperiments on Isolated Fragments

There are times where a SD fragment from a DNA molecule is isolated and placed in a vector so it can be examined as a smaller subproblem. Typically, there will be many sites for other enzymes of interest within the fragment, but there will be insufficient information to resolve the map within the SD fragment. The subexperiment will involve additional enzymes, but will always have enzymes in common with the parent map whose sites need to be resolved. The solution map for the subexperiment can be generated independently, thus also constraining the common sites.

Figure 4 shows the sites for common enzymes within the subexperiment fragment and the parent map. The ends of the isolated fragment in the parent map are at sites e_i and e_{i+m}; it is inserted at site f_j, being cut by the same enzyme, and extends to site f_{j+m} in the subexperiment. To simplify this figure, only common enzymes are shown; there typically will be other digestions for either map that do not contribute mutual constraints.

Two sets of mutual constraints for the common enzymes result:

$$e_i = f_j, \quad \ldots, \quad e_{i+m} = f_{j+m}, \tag{1}$$

$$U_i = V_j, \quad \ldots, \quad U_{i+m} = V_{j+m}. \tag{2}$$

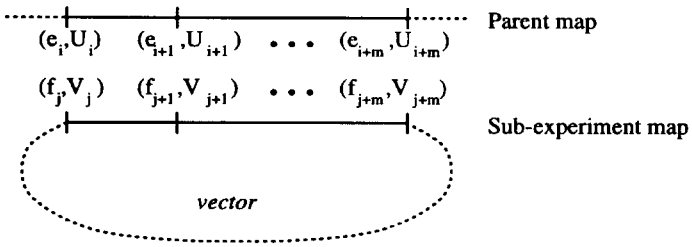


FIGURE 4. Common sites in subexperiment.

Constraints (1) specify the same site configuration for common enzymes. Constraints (2) ensure that the relative positions of the sites are mutually compatible.

Since we suppress the generation of symmetric solutions, we must consider the reflective symmetry of the subexperiment maps. This is done by reversing the site ordering and reversing the sign of the site variables. The two alternative sets of constraints for a reversed subexperiment map are

$$e_i = f_{j+m}, \quad \ldots, \quad e_{i+m} = f_j, \tag{3}$$

$$U_i = -V_{j+m}, \quad \ldots, \quad U_{i+m} = -V_j. \tag{4}$$

We denote the mutual consistency check of constraints (1) $\wedge$ (2) $\vee$ (3) $\wedge$ (4) by the operator $\bullet$. A parent map can only succeed if it is consistent with at least one solution map or reversed map from each subexperiment.

Consider a parent map M, a subexperiment map $S_1 = \{m_1, m_2, \ldots\}$, i.e., its solution set, and another subexperiment map $S_2 = \{n_1, n_2, \ldots\}$. Finding subexperiment maps that confirm M is represented by

$$M \bullet choice(S_1) \bullet choice(S_2),$$

where $choice(S)$ backtracks through all choices from set S. The choice operator is required because the constraints imposed by (2) or (4) may propagate to sites beyond the inserted subexperiment fragment. Thus, confirming a parent map involves a backtracking search through the cross product of the solution sets for the subexperiments. A parent map may have more than one confirming combination of subexperiment maps.

Were we to apply the $\bullet$ operator for each inserted subexperiment fragment, these constraints from the subexperiment would strongly prune the search space. However, the choice operator becomes a branching factor and maps with multiple confirming subexperiment maps would be returned as distinct solutions.

To overcome this problem, at first we only apply constraints from the site configuration, that is, constraints (1) $\vee$ (3). The compromise operator $\circ$ is represented by

$$M \circ first(S_1) \circ first(S_2),$$

where $first(E)$ finds the first element in E that satisfies the $\circ$ operator. The choice operator becomes unnecessary because constraints (1) are local constraints. There can be no other site configuration for the common sites. The $\circ$ operator is weaker than $\bullet$ and hence is complete. To ensure soundness, we apply the $\bullet$ operator only at the very end of each subexperiment map generation.

We know of no other work that takes this approach, and the optimizations for subexperiments are entirely due to us. The weaker site configuration constraint is most effective in cutting down the search space and execution time. There tends to be a surprisingly small number of site configurations in subexperiments. The results suggest that for subexperiments with 8 sites and up to 50 solutions, the number of site configurations is usually one and hardly ever more than two.

7.2. Pipelining

Where additional digestions are performed on the same DNA, merging is again performed using mutual constraints for the common sites. As an example, a biologist could perform SDs and DDs with enzymes *a* and *b*—Digest[a], Digest[b], and Digest[a, b]—and find there are many maps satisfying the constraints. To further reduce the solution set, additional digests with enzyme *c* could be performed—Digest[c], Digest[a, c], and Digest[b, c].

Performing map generation in stages rather than permuting all the digests simultaneously is usually significantly more efficient [9]. We denote the *simultaneous* permutation of fragments for the *a*, *b*, and *ab* digests (as presented in Section 3) by

[a, b, ab]

and pipelining for this example by

[a, b, ab] → [c, ac, bc].

The ability to partially instantiate logical variables provides a convenient mechanism to use the solution maps obtained for digests *a*, *b*, and *ab* as starting points for pipelined map generation with the new data. The fragment orders from the *a*, *b*, and *ab* digests are already known. We only need permute fragments from the new digests *c*, *ac*, and *bc*.

Say there are two solutions m1 and m2 over the domain Digest[a], Digest[b], and Digest[a, b]:

m1 = Site:	[a, b, a, b]	m2 = Site:	[a, b, b, a]
Digest[a]:	[a1, a2]	Digest[a]:	[a1, a2]
Digest[b]:	[b1, b2]	Digest[b]:	[b2, b1]
Digest[ab]:	[ab1, ab2, ab3, ab4]	Digest[ab]:	[ab1, ab3, ab2, ab4]

To cater for the new digestions Digest[c], Digest[a, c], and Digest[b, c], we first expand the domain to include the new digests:

m1′ = Site:	[a, b, a, b]	m2′ = Site:	[a, b, b, a]
Digest[a]:	[a1, a2]	Digest[a]:	[a1, a2]
Digest[b]:	[b1, b2]	Digest[b]:	[b2, b1]
Digest[a, b]:	[ab1, ab2, ab3, ab4]	Digest[a, b]:	[ab1, ab3, ab2, ab4]
Digest[c]:	_	Digest[c]:	_
Digest[a, c]:	_	Digest[a, c]:	_
Digest[b, c]:	_	Digest[b, c]:	_

The map generator is given m1′ as an initial value, and the same is done for m2′ when map generation with m1′ is completed.

Indeed, it is usually better to use the following three-stage pipeline instead of the above [6]:

[a, b, ab] → [c, ac] → [bc].

Notice, not all applications of the operators are useful. For example, [a, b] or [a] → [b] would generate all possible permutations of the *a* and *b* fragments but with no mutual constraints for pruning.

7.3. Cross Multiplication

Another path that biologists follow is to divide the problem into subproblems involving a smaller number of enzymes and digestions, find maps for the subproblems, and then merge the submaps to form the final map, again using mutual constraints for the digests in common.

We use the cross multiplication operator × to denote merging of the solution sets of such submaps. The result is a solution set in the larger domain involving the union of all the enzymes and digests in the submaps. Application of cross multiplication is appropriate where the mapping problem involves many digestions, each with a relatively large number of fragments, where the chosen subproblems have very few common enzymes or common digests. The cross multiplication operator will merge submaps where the common digests are compatible. The optimization we use in this merge is to check only for compatible site orders using the site and fragment configurations for common enzymes.

The example in the previous section involves six digestions and three enzymes. Say the problem were divided into subproblems involving two enzymes: [a, b, ab] and [b, c, bc]. Moreover, say the first two subproblems were mapped: the submap for [a, b, ab] had the solution set {m1, m2} and the submap for [b, c, bc] had the solution set {n1}:

m1 =	Site:	[a, b, a, b]	m2 =	Site:	[a, b, b, a]
	Digest[a]:	[a1, a2]		Digest[a]:	[a1, a2]
	Digest[b]:	[b1, b2]		Digest[b]:	[b2, b1]
	Digest[ab]:	[ab1, ab2, ab3, ab4]		Digest[ab]:	[ab1, ab3, ab2, ab4]

n1 =	Site:	[b, c, b, c]
	Digest[b]:	[b1, b2]
	Digest[c]:	[c1, c2]
	Digest[bc]:	[bc1, bc2, bc3, bc4]

Notice the fragment order for m2 is incompatible with that of n1 because the *b* digest fragment orders are different. Cross multiplication would eliminate that pairing due to an incompatible fragment configuration. This leads to the optimization we use for cross multiplication: only site and fragment ordering constraints are checked.

Merging m1 with n1 gives

Site:	[a, b, (a, c), b, c]
Digest[a]:	[a1, a2]
Digest[b]:	[b1, b2]
Digest[c]:	[c1, c2]
Digest[ab]:	[ab1, ab2, ab3, ab4]
Digest[bc]:	[bc1, bc2, bc3, bc4]

The partial order [a, b, (a, c), b, c] is obtained by aligning sites for common enzymes; enzyme is *b* in this example:

```
m1:  a b a b
       |   |
n1:  b c b c
```

That is, the site configurations of the submaps must be compatible. For partial orders, the atoms within parentheses can appear in any order, and either or both may be valid solutions. When partial orders need to be resolved, all appropriate site configurations are considered.

Notice that only ordering information has been applied above, from the site and fragment configurations. To ensure the fragment length constraints are mutually consistent, a further step is performed:

$$[a, b, ab] \times [b, c, bc] \rightarrow [\ ].$$

Here, cross multiplication provides a starting point for the map generator. The pipeline step confirms the lengths. The time for this extra confirming step is small compared to the simultaneous generation from all fragments. From experience, the extra step is almost always worthwhile. If no other maps are to be merged, the extra step must be performed to maintain soundness for cross multiplication.

The initial investigation of pipelining and cross multiplication was done with colleagues [9, 10]. Our subsequent work [6] shows that although pipelining is usually significantly faster, the relative efficiency of these operators is data-dependent. The optimization for cross multiplication is due to us.

8. INTEGRATING OTHER EXPERIMENTAL CONSTRAINTS

In this section, we give constraints for other experimental techniques. We aim here to show that such techniques fit in well with constraint logic programming. The bottom layer for permutation generation requires augmentation with typical recursive code to handle the additional cases.

8.1. Probes

A probe is a short piece of radioactive DNA that is used to bind, or hybridize, with target DNA having a complementary pattern. A probe is short enough that it can be considered a single point in the DNA. We consider the case where a probe only hybridizes the target DNA at a unique location. This follows usual experimental practice where the data are usually discarded if hybridization occurs at more than one location. Digestions are performed on the hybridized DNA as usual, and for each digestion one of the fragments will be identified as being radioactive, that is, containing the probe.

A probe site is a unique location in the DNA, or on the resulting map. For the same probe, hybridized fragments from different digestions are constrained to occupy the same locality. In CLP(R), we can represent a probe very simply by a site variable V_p. In placing a hybridized fragment between the sites V_j and V_k, where $j < k$, we only have to introduce the constraint

$$V_j \leq V_p \leq V_k, \tag{5}$$

or if site k crosses the circular boundary,

$$V_j \leq V_p \leq V_k + C. \tag{6}$$

With slight modifications in the map generator, we can force the hybridized fragments from different digestions to be placed within the same vicinity. This is achieved by issuing the following constraint when placing a fragment up to a new site V_i before the hybridized fragment:

$$V_i \leq V_p. \tag{7}$$

Eventually, a hybridized fragment for one of the digests will be placed and constraints (5) or (6) will be issued. Thereafter, constraint (7) cannot be issued again (without backtracking) for site V_p. In essence, the placement of the first hybridized fragment forces the placement of the corresponding hybridized fragments for every other digestion immediately afterwards. This reduces the search space significantly.

8.2. End-Labelling

End-labelling is used to identify fragments that include part of the vector. In effect, one probe is attached to the vector near the site where the DNA is inserted. If both ends are labelled, two different probes are used. In Figure 5, the insertion was performed at the b site. p_1 is the label near one end of the vector and p_2 is the other. When digested with another enzyme, e in this case, the end fragments e1 and e2 will each contain part of the vector. These are called end fragments and have been labelled with p_1 and p_2, respectively.

The identification of end fragments limits the locations at which e1 and e4 can be placed, and there are two possibilities for this example:

Digest[e]: [e1,_,_,e4]
Digest[e]: [e4,_,_,e1]

We can call the map generator with each of the assignments, thus reducing the search space.

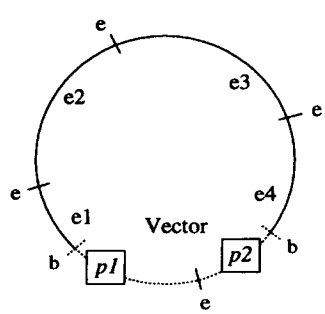


FIGURE 5. Digestion with enzyme e, fragment inserted at b site.

8.3. Partial Digestion and End-Labelling Constraints

A less common technique used by biologists is to stop the digestion by a single enzyme before completion, a *partial* digestion (PD). The result is a collection of *partially digested* fragments, that is, fragments that may contain uncleaved sites for the cutting enzyme. Recall that all digestions are performed in a solution of very many cloned DNA molecules. Any of the clones could be cleaved at any subset of all of its sites. The number of different PD fragments is potentially $n \times (n-1)$, where n is the number of enzyme sites. This number is usually too big for mapping purposes. To overcome the potentially large number of different fragments from a partial digestion, biologists also use end-labelling.

Partial digestion with end-labelling is performed as two single-digest experiments, one for each end. First, one end of the vector is labelled and a partial digestion is performed. Only those fragments that are labelled are measured. These fragments all have a common end or reference point. This is illustrated by the fragments f1, f2, and f3 in Figure 6. The same experiment is repeated with a different label at the other end.

Ideally, fragments from the reference point to all sites would be obtained. However, in practice, long fragments cannot be measured with sufficient accuracy due to the existing experimental technique. Hence, we deliberately omitted the large fragment from s_1 to s_1. More generally, measurements for fragments to sites in the middle of larger maps would not be available either.

From the two experiments, we obtain two sorted sets of fragments each containing three fragments. All the fragments share the reference site (V_1, e_1). The first set contains fragments extending in one direction from the reference site to the sites V_2, V_3, and V_4; the second set contains fragments extending in the other direction.

Let L_1 denote the lower bound of f1 and H_1 its upper bound, similarly for L_2 and H_2, etc. The constraints imposed by these fragments are simply

$$L_1 \le V_2 - V_1 \le H_1, \qquad L_4 \le V_1 + C - V_4 \le H_4,$$
$$L_2 \le V_3 - V_1 \le H_2, \qquad L_5 \le V_1 + C - V_3 \le H_5,$$
$$L_3 \le V_4 - V_1 \le H_3, \qquad L_5 \le V_1 + C - V_2 \le H_6.$$

The sorting of fragments in this manner considerably simplifies the search. However, it is only suitable for small DNA molecules, and not all sites are guaranteed to be cut.

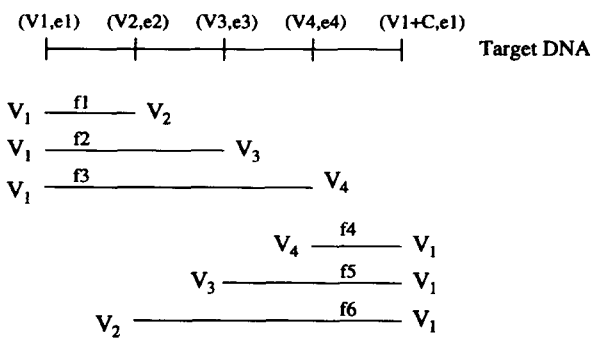


FIGURE 6. Partial digestion with end-labelling.

9. CONCLUSION

We have used CLP(R) to provide a platform for restriction mapping. The system is sound and complete, finding all maps within the bounds placed on fragments. Our implementation covers the standard single- and double-digests, the merging of submaps and allowance for vectors. We also described the constraints and partial instantiation necessary for end-labelling, probes, and partial digestion with end-labelling that can be naturally incorporated in our framework.

The exponential growth in search space is the determining factor in execution time and number of consistent solutions obtained in restriction mapping. Our experience shows that map generation combined with subexperiments, pipelining, or cross multiplication significantly reduces the search space and time taken [6]. Our judgment is that as new techniques are developed by biologists, it must be relatively easy to add code for them. We have demonstrated this while building the system thus far, and have illustrated the relatively simple constraints that might be added. The ease of adaptation of our implementation to additional techniques demonstrates the superior expressive power of the logic programming framework. The prospect of adding more techniques to our system would be daunting were it written in a conventional language.

We initially chose CLP(R) as a prototype language. However, we soon realized that despite the relatively high language overhead of CLP(R), we were able to solve mapping problems that were too large for our previous programs written in C [5, 10]. In part, this was due to the optimizations we developed. However, the ability to combine data from different techniques also makes a significant contribution, subexperiments in particular.

We started with a generalized permutation map generator for an arbitrary number of enzymes and digestions, and added more code for vector sites, etc. During this process, very few changes were made to the permutation generator that forms the core of our implementation. The permutation map generator, excluding comments, is about 35 lines. There are about 900 lines of code and another 900 of comments.

Our implementation is demonstrative of the declarative power of logic programming. We make full and extensive use of logical variables, partial instantiation, and backtrack search goal reduction. The incremental linear programming solver of CLP(R), which solves the restriction mapping constraints automatically, is indispensable. The mapping constraints are simply stated as program goals. This contrasts markedly with the separation theory solver in [10], where the constraint module is the largest part of the program.

REFERENCES

1. Allison, L., Dix, T. I., and Yee, C. N., Shortest Path and Closure Algorithms for Banded Matrices, *Information Processing Letters* 40:317–322 (1991).
2. Allison, L. and Yee, C. N., Restriction Sites Mapping Is in Separation Theory, *Computer Applications in Biosciences* 4:91–101 (1988).
3. Bellon, B., Construction of Restriction Maps, *Computer Applications in Biosciences* 4:111–115 (1988).
4. Dix, T. I. and Ho-Stuart, C. J., Constraint Checking for Circular Restriction Site Mapping, in: *Twenty-Fifth Annual Hawaii International Conference on Systems Sciences*, IEEE Press, New York, 1992, pp. 635–642.

5. Dix, T. I. and Kieronska, D. H., Errors between Sites in Restriction Site Mapping, *Computer Applications in Biosciences* 4:117–123 (1988).
6. Dix, T. I. and Yee, C. N., A Restriction Mapping Engine Using Constraint Logic Programming, in: *Second International Conference on Intelligent Systems for Molecular Biology*, AAAI Press, Menlo Park, CA, 1994, pp. 112–120.
7. Fitch, W. M., Smith, T. F., and Ralph, W. W., Mapping the Order of DNA Restriction Fragments, *Gene* 22:19–29 (1983).
8. Goldstein, L. and Waterman, M. S., Mapping DNA by Stochastic Relaxation, *Advances in Applied Mathematics* 8:194–207 (1987).
9. Ho, S. T. S., Allison, L., and Yee, C. N., Restriction Site Mapping for Three or More Enzymes, *Computer Applications in Biosciences* 6:195–204 (1990).
10. Ho, S. T. S., Allison, L., Yee, C. N., and Dix, T. I., Constraint Checking for Restriction Site Mapping, in *Twenty-Fourth Annual Hawaii International Conference on Systems Sciences*, IEEE Press, New York, 1991, pp. 605–614.
11. Jaffar, J. and Lassez, J.-L., Constraint Logic Programming, in: *Proceedings 14th ACM Symposium on Principles of Programming Languages*, 1987, pp. 111–119.
12. Jaffar, J., Michaylov, S., Stuckey, P. J., and Yap, R. H. C., The CLP(R) Language and System, *ACM Transactions on Programming Languages and Systems* 14(3):339–395 (1992).
13. Krawczak, M., Algorithm for Restriction Site Mapping of DNA Molecules, *Proceedings of National Academy of Science USA* 85:7298–7301 (1988).
14. Pearson, W. R., Automatic Construction of Restriction Site Maps, *Nucleic Acids Research* 10:217–227 (1982).
15. Stefik, M., Inferring DNA Structure from Segmentation Data, *Artificial Intelligence* 11:85–114 (1978).
16. Yap, R. H. C., A Constraint Logic Programming Framework for Constructing DNA Restriction Maps, *Artificial Intelligence in Medicine* 5:447–464 (1993).
17. Zehetner, G., Frischauf, A., and Lehrach, H., Approaches to Restriction Map Determination, in: M. J. Bishop and C. J. Rawlings (eds.), *Nucleic Acid and Protein Sequence Analysis, A Practical Approach*, IRL Press, Oxford, 1987, pp. 147–164.
18. Zehetner, G. and Lehrach, H., A Computer Program Package for Restriction Map Analysis and Manipulation, *Nucleic Acids Research* 14:335–349 (1986).